Table of Contents

SPRING BOOT TUTORIAL

WHY TO LEARN SPRING BOOT

Spring Boot provides a great range of features and benefits:

- **Flexibility**– Spring Boot provides multiple flexible ways to configure Java Beans, XML configurations, and Database Transactions.
- **Powerful Batch Processing**– Spring Boot provides a strong batch mechanism.
- **Microservices Support**– Micro Service is an architecture that allows the developers to develop and deploy services independently. Each service running has its own process, and this achieves the lightweight model to support business applications. Spring Boot provides mechanism to develop and test microservices easily.
- **Auto Configuration**– In Spring Boot, everything is auto configured; no manual configurations are needed.
- **Annotation Based**– In Spring Boot, we can create a running application with very few annotations.
- **Eases dependency management**– Spring Boot provides many starters as per need like for web, for database etc to handle dependencies effectively. A starter project provides dependency management for corresponding functionalities.
- **Embedded Servlet Container**– Spring Boot provides an embedded servlet container(jetty) which can be changed as well. This container is very useful while testing the application. We can test all functionalities without deploying the spring boot application on any external application container.

APPLICATIONS OF SPRING BOOT

- POJO Based - Spring Boot enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server, but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
- **Modular** - Spring Boot is modular by nature. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
- **Integration with existing frameworks** - Spring Boot does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
- **Testablity** - Testing an application written with Spring Boot is very easy because environment-dependent code is moved into this framework. Furthermore, by using Java Bean style POJOs, it becomes easier to use dependency injection for injecting test data.
- Web MVC - Spring Boot's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
- **Central Exception Handling** - Spring Boot provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- **Lightweight** - Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.

- **Transaction management** - Spring Boot provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

GOALS OF SPRING BOOT

- **Simpler Configuration**– To avoid complex XML configuration in Spring. With very few annotations, we can achieve the same configurations.
- **Ease of Development**– To develop a production ready Spring applications in an easier way.
- **Reduce Development Efforts**– To reduce the development time and run the application independently.
- **Easy Deployment**– Offer an easier way of getting started with the application by providing an embedded application container to test the application locally.

WHAT IS SPRING BOOT

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

Advantages

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

Goals

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

Why Spring Boot

- It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
- It provides a powerful batch processing and manages REST endpoints.
- In Spring Boot, everything is auto configured; no manual configurations are needed.
- It offers annotation-based spring application
- Eases dependency management
- It includes Embedded Servlet Container

How does it work

- Spring Boot automatically configures your application based on the dependencies you have added to the project by using **@EnableAutoConfiguration** annotation. For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

- The entry point of the spring boot application is the class contains the annotation **@SpringBootApplication** and the main method.
- Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation and it scans @Component, @Service, @Repository, @Controller

Spring Boot Starters

- Managing dependencies in large-scale Java projects can be complex. **Spring Boot simplifies this** by providing a curated set of dependencies through **starter packages**.
- For example, if you want to use Spring and JPA for database access, it is sufficient if you include **spring-boot-starter-data-jpa** dependency in your project.
- Note that all Spring Boot starters follow the same naming pattern **spring-boot-starter-** *, where * indicates that it is a type of the application.

Examples:

Starter	**Purpose**
spring-boot-starter-web	Build web applications (REST, MVC)
spring-boot-starter-data-jpa	Integrate Spring Data JPA for database access
spring-boot-starter-security	Add Spring Security features
spring-boot-starter-test	Include testing libraries (JUnit, Mockito, etc.)
spring-boot-starter-thymeleaf	Use Thymeleaf as a templating engine

Auto Configuration

- Spring Boot's **Auto Configuration** feature automatically sets up your application based on the dependencies present in the classpath. This reduces the need for manual configuration and speeds up development.
- Spring Boot uses spring.factories to load configuration classes.
- These classes contain conditional logic (@ConditionalOnClass, @ConditionalOnMissingBean, etc.) to apply configurations only when needed.

Spring Boot Application

- The entry point of the Spring Boot Application is the class contains @**SpringBootApplication** annotation.
- This class should have the main method to run the Spring Boot application. **@SpringBootApplication** annotation includes Auto-Configuration, Component Scan, and Spring Boot Configuration.

Component Scan

- Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the **@ComponentScan** annotation for your class file to scan your components added in your project.

HOW TO CREATE A SPRING BOOT APPLICATION

1. Spring Initializr (Web Interface)

- Visit start.spring.io
- Choose project type, language, metadata, dependencies
- Download and extract the ZIP file

2. Spring Initializr via IDE

- IntelliJ IDEA: File → New → Project → Spring Initializr
- Eclipse (STS): File → New → Spring Starter Project
- VS Code: Use Spring Boot Extension Pack

3. Command Line (cURL or HTTPie)

- Use terminal to send request to Spring Initializr
- Example: curl or http command to generate ZIP

4. Spring Boot CLI

- Install Spring Boot CLI
- Use spring init command to generate project

5. Manual Setup

- Create Maven/Gradle project manually
- Add Spring Boot dependencies
- Create main class with @SpringBootApplication

DEPENDENCY MANAGEMENT

Spring Boot simplifies dependency management by providing a curated list of dependencies for each release, ensuring compatibility among libraries. Developers do not need to specify versions manually in pom.xml or build.gradle because Spring Boot automatically manages the correct versions for all included dependencies. When you upgrade the Spring Boot version, all managed dependencies are also upgraded to their compatible versions, making maintenance easier. Although manual version specification is allowed, it is generally not recommended by the Spring Boot team, as overriding versions can lead to conflicts and break compatibility.

Maven Dependency

For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies. For this, simply we can inherit the starter parent in our **pom.xml** file as shown below.

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>3.5.6</version>
</parent>
```

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number. Observe the code given below.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

Gradle Dependency

We can import the Spring Boot Starters dependencies directly into **build.gradle** file. We do not need Spring Boot start Parent dependency like Maven for Gradle. Observe the code given below.

```
buildscript {
   ext {
      springBootVersion = '3.5.6'
   }
   repositories {
      mavenCentral()
   }
   dependencies {
      classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
   }
}
```

Similarly, in Gradle, we need not specify the Spring Boot version number for dependencies. Spring Boot automatically configures the dependency based on the version.

```
dependencies {
   compile('org.springframework.boot:spring-boot-starter-web')
}
```

BEANS & DEPENDENCY INJECTION

Bean

A **bean** is an object that is managed by the Spring container. Beans are created automatically when you annotate classes with Spring stereotypes like @Component, @Service, @Repository, or @Controller.

Dependency Injection (DI)

Dependency Injection is a design pattern where the control of creating and managing object dependencies is transferred from the application code to the Spring framework. Instead of instantiating objects manually, Spring injects them where needed, making the code more modular, testable, and maintainable.

Types of injection

1. Constructor Injection (Recommended)

Dependencies are provided through the class constructor. This is the most preferred method in Spring Boot because it ensures immutability and makes the class easier to test.

Advantages:

- ❖ Promotes immutability (fields can be final)
- ❖ Easier to write unit tests (no need for reflection or setters)
- ❖ Ensures all required dependencies are provided at object creation

```java
@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }
}
```

2. Setter Injection

Dependencies are injected via public setter methods. This allows optional dependencies and flexibility in configuration.

Advantages:

- ❖ Good for optional dependencies
- ❖ Can be modified after object creation

```java
@Service
public class NotificationService {
    private EmailService emailService;

    @Autowired
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }
}
```

3. Field Injection (Not Recommended)

Dependencies are injected directly into fields using the @Autowired annotation. While convenient, it's discouraged due to poor testability and tight coupling.

Advantages:

- ❖ Less boilerplate code
- ❖ Quick and easy for small projects

Disadvantages:

- ❖ Harder to test (requires reflection or Spring context)
- ❖ Breaks encapsulation
- ❖ Not suitable for large-scale or production-grade applications

```java
@Service
public class PaymentService {
    @Autowired
    private PaymentGateway paymentGateway;
}
```

Summary Table

Injection Type	Recommended	Testability	Immutability	Use Case
Constructor	☑ Yes	☑ High	☑ Yes	Required dependencies
Setter	⚠ Sometimes	☑ Moderate	❌ No	Optional dependencies
Field	❌ No	❌ Low	❌ No	Quick setup (not for prod)

RUNNERS

Spring Boot provides two interfaces — CommandLineRunner and ApplicationRunner — to execute code after the application context is loaded and before the Spring Boot application is fully started. These are useful for tasks like initializing data, running setup logic, or triggering background jobs.

CommandLineRunner

Executes logic with raw command-line arguments passed as a String[].

```java
@Component
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("CommandLineRunner executed with args: " +
Arrays.toString(args));
    }
}
```

ApplicationRunner

Executes logic using **ApplicationArguments**, which provides structured access to command-line arguments.

```java
@Component
public class MyApplicationRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("ApplicationRunner executed with options: " +
args.getOptionNames());
    }
}
``
```

USE CASES FOR RUNNERS

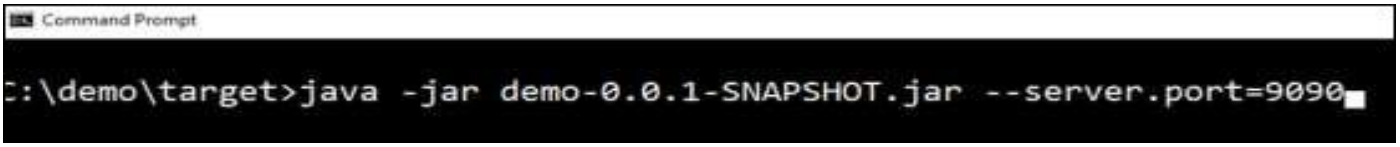
- Preloading data into the database
- Triggering background jobs
- Logging startup information
- Validating configurations

APPLICATION PROPERTIES

Command Line Properties

Spring Boot application converts the command line properties into Spring Boot Environment properties. Command line properties take precedence over the other property sources. By default, Spring Boot uses the 8080-port number to start the Tomcat. Using below steps change the port number by using command line properties.

- Step 1 – After creating an executable JAR file, run it by using the command java jar <JARFILE>.
- Step 2 – Use the command given in the screenshot given below to change the port number for Spring Boot application by using command line properties



Note – You can provide more than one application properties by using the delimiter.

Properties File

Properties files are used to keep N number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the application.properties file under the classpath.

The **application.properties** file is in the **src/main/resources** directory.

The code for sample application.properties file is given below

```
server.port = 9090
spring.application.name = demoservice
```

Note that in the code shown above the Spring Boot application demoservice starts on the port 9090.

YAML File

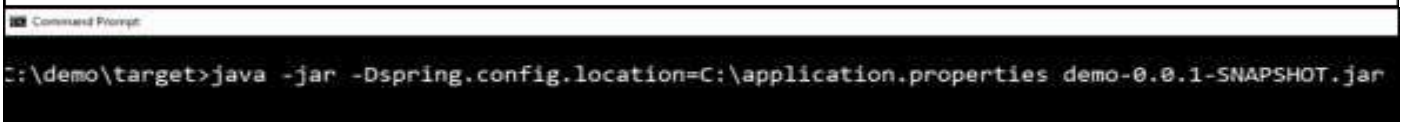
Spring Boot supports YAML based properties configurations to run the application. Instead of **application.properties**, we can use **application.yml** file. This YAML file also should be kept inside the classpath. The sample **application.yml** file is given below

```yaml
spring:
   application:
      name: demoservice
   server:
port: 9090
```

Externalized Properties

Instead of keeping the properties file under classpath, we can keep the properties in different location or path. While running the JAR file, we can specify the properties file path. You can use the following command to specify the location of properties file while running the JAR.

```
-Dspring.config.location = C:\application.properties
```



Use of @Value Annotation

The @Value annotation is used to read the environment or application property value in Java code. Look at the following example that shows the syntax to read the **spring.application.name** property value in Java variable by using @Value annotation.

```
@Value("${spring.application.name}")
```

Observe the code given below for a better understanding –

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {
   @Value("${spring.application.name}")
   private String name;
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
   @RequestMapping(value = "/")
   public String name() {
      return name;
   }
}
```

Note – If the property is not found while running the application, Spring Boot throws the Illegal Argument exception as **Could not resolve placeholder 'spring.application.name' in value "${spring.application.name}"**.

Spring Boot Active Profile

Spring Boot supports different properties based on the Spring active profile. For example, we can keep two separate files for development and production to run the Spring Boot application.

SPRING ACTIVE PROFILE IN application.properties

Let us understand how to have Spring active profile in application.properties. By default, application. properties will be used to run the Spring Boot application. If you want to use profile-based properties, we can keep separate properties file for each.

- Default file: application.properties
- Profile-specific files:
  - application-dev.properties
  - application-prod.properties

CONFIGURATION/AUTO-CONFIGURATION

SPRING BOOT CONFIGURATION

Although a Spring/Spring Boot application can be configured in an XML document, it is generally recommended to configure Spring Boot applications with a single file with **@Configuration** annotation.

@Component Annotation

In Spring Boot, the **@Component** annotation declares a class as a Spring component. This signals to Spring that the class should be managed by the Spring container, meaning it will be automatically created, initialized, and potentially injected into other components.

@Configuration Annotation

In Spring Boot, the **@Configuration** annotation is used to indicate that a class is a source of bean definitions. It's a core annotation from the Spring Framework, not specific to Spring Boot. Classes marked with **@Configuration** can define beans using the **@Bean** annotation on methods within the class. These beans are then managed by the Spring container. **@Configuration** promotes a Java-based configuration approach instead of relying solely on XML configuration files.

@SpringBootApplication Annotation

This is a meta-annotation that combines @EnableAutoConfiguration, @ComponentScan, and @Configuration. It's often the only annotation you need to bootstrap a Spring Boot application.

@ComponentScan Annotation

In Spring Boot, the @ComponentScan annotation is used to instruct Spring to scan the specified packages and their sub-packages for components (classes annotated with @Component, @Service, @Repository, @Controller, etc.) and register them as beans in the Spring application context.

By default, if you don't explicitly use @ComponentScan, Spring Boot automatically scans the package where your main application class resides and its sub-packages. If your components are in a different package structure, you can use base Packages attribute of @ComponentScan to specify the packages to scan. You can specify multiple packages using an array in base Packages.

SPRING BOOT AUTOCONFIGURATION

Auto-Configuration is a powerful feature of Spring Boot that automatically configures your application based on the dependencies present in the classpath. It uses @EnableAutoConfiguration (internally part of @SpringBootApplication) and a set of conditional annotations like @ConditionalOnClass, @ConditionalOnMissingBean, etc.

@SpringBootApplication or @EnableAutoConfiguration Annotation

These annotations trigger the autoconfiguration process. Typically, you annotate your main application class with @SpringBootApplication, which includes @EnableAutoConfiguration. Additional packages can be configured using the @AutoConfigurationPackage annotation.

- **Scanning the Classpath** – Spring Boot scans the classpath for libraries and frameworks.
- **Conditional Configuration** – Based on the detected dependencies, Spring Boot applies relevant configurations. This is done using @Conditional annotations, such as:
  - @ConditionalOnClass– Applies configuration if a specific class is present on the classpath.
  - @ConditionalOnMissingBean– Applies configuration if a specific bean is not already defined.
  - @ConditionalOnProperty– Applies configuration based on property values.
- **Bean Creation and Configuration** – Spring Boot creates and configures beans based on the applied autoconfigurations.

Example:

- If you add the **spring-boot-starter-web** dependency to your project, Spring Boot automatically configures a Tomcat web server and other necessary components for a web application.
- If you add the **spring-boot-starter-data-jpa** dependency, Spring Boot automatically configures a data source, entity manager, and other components required for JPA-based data access.

Benefits of Spring Boot Autoconfiguration

- **Reduced Boilerplate Code**– Autoconfiguration eliminates the need for manual configuration of common Spring components.
- **Faster Development**– You can quickly create Spring Boot applications without having to write extensive configuration code.
- **Improved Productivity**– Autoconfiguration allows you to focus on the core business logic of your application.
- **Customization**– You can customize the autoconfiguration behaviour by following means:
  - Excluding specific autoconfiguration classes using the exclude attribute of **@EnableAutoConfiguration**.
  - Defining your own beans to override the auto-configured ones.

- Using property files or environment variables to modify the default configurations.

ANNOTATIONS

Annotations are metadata tags in Java that provide information to the compiler and runtime environment. In Spring Boot, annotations are heavily used to simplify configuration, define beans, enable features, and manage application behaviour declaratively.

Annotations

@RestController

@RestController marks a class as controller for RESTful web requests. It combines the functionality of **@Controller and @ResponseBody**.

- **@Controller**– Marks the class as a Spring MVC controller, responsible for handling incoming HTTP requests.
- **@ResponseBody**– Tells Spring to serialize the return value of the controller's method directly into the HTTP response body, typically in formats like JSON or XML.

HTTP Related Annotations

@RequestMapping

@RequestMapping annotation is used to map web requests to specific handler methods in a controller class. It acts as a bridge between the incoming HTTP requests and the methods responsible for handling them. You can specify the HTTP method (GET, POST, PUT, DELETE, etc.) that the method handles using the method attribute.

With latest version of Spring Boot, the preferred way is shortcut annotations instead of @RequestMapping

- @GetMapping– Shortcut for @RequestMapping(method = RequestMethod.GET)
- @PostMapping– Shortcut for @RequestMapping(method = RequestMethod.POST)
- @PutMapping– Shortcut for @RequestMapping(method = RequestMethod.PUT)
- @DeleteMapping– Shortcut for @RequestMapping(method = RequestMethod.DELETE)
- @PatchMapping– Shortcut for @RequestMapping(method = RequestMethod.PATCH)

@PathVariable

@PathVariable is a Spring annotation used to extract values from the URI path and bind them to method parameters in a controller. It's commonly used in REST APIs to capture dynamic values from the URL.

It allows you to create dynamic endpoints where parts of the URL can vary based on user input or resource identifiers.

```java
@GetMapping("/users/{userId}/orders/{orderId}")
public String getOrderDetails(@PathVariable int userId, @PathVariable int orderId) {
    return "User ID: " + userId + ", Order ID: " + orderId;
}

```

@RequestParam

@RequestParambinds a method parameter to a request parameter.

@RequestBody

@RequestBodybinds a method parameter to the body of the HTTP request.

@ModelAttribute

@ModelAttribute binds a method parameter to a model attribute. In Spring Boot, a model variable is an interface that holds data to be passed from a controller to a view. This interface Model is in org.springframework.ui package.

JPA Related Annotations

@Repository

@Repository marks a class as a **data access object (DAO).** This class is responsible for CRUD operations on your database.

@Entity

@Entitymarks a class as a JPA entity.

@Id

@Idspecifies the primary key of an entity.

@GeneratedValue

@GeneratedValue specifies how the primary key should be generated.

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

@Column

Column maps a field to a database column.

@Transactional

@Transactional is a Spring annotation used to manage transactions declaratively. It ensures that a method executes within a database transaction — meaning all operations inside the method are treated as a single unit of work. If any part fails, the entire transaction is rolled back. Add @EnableTransactionManagement to your main application class.

Other important annotations

@Value

This annotation is used to assign default values to variables and method arguments. It can also be used to inject a property value from a properties file or environment variable.

@EnableCaching

@EnableCaching enables caching support. When you add @EnableCaching to a configuration class, Spring scans the application for methods annotated with caching - related annotations (such as @Cacheable, @CachePut, @CacheEvict, or @Caching).

@EnableAsync

@EnableAsync enables support for asynchronous method execution. Methods annotated with @Async will run in separate threads, allowing the caller to proceed without waiting for the method to complete.

LOGGING

Logging is a critical part of any application for monitoring, debugging, and troubleshooting. Spring Boot uses Apache Commons Logging as a facade and supports popular logging frameworks like:

- Logback (default)
- Log4j2
- Java Util Logging (JUL)

Log Format

```
[2025-09-28T15:56:08Z] [org.apache.juli.logging.DirectJDKLog] [main] [168] [INFO ]
Starting ProtocolHandler ["http-nio-8080"]
[2025-09-28T15:56:08Z] [org.springframework.boot.web.embedded.tomcat.TomcatWebServer]
[main] [243] [INFO ] Tomcat started on port 8080 (http) with context path '/'
[2025-09-28T15:56:08Z] [org.springframework.boot.StartupInfoLogger] [main] [59] [INFO ]
Started DemoApplication in 2.077 seconds (process running for 2.568)
```

Generic Log Format Breakdown

Part	**Description**
[2025-09-28T15:56:08Z]	**Timestamp** – When the log entry was created (ISO 8601 format)
[org.apache.juli.logging.DirectJDKLog]	**Logger Name** – Class or component generating the log
[main]	**Thread Name** – Thread executing the log statement
[168]	**Line Number or Log ID** – Often represents internal tracking or line
[INFO]	**Log Level** – Severity of the message (INFO, DEBUG, ERROR, etc.)
Starting ProtocolHandler ["http-nio-8080"]	**Log Message** – Actual content of the log

Types of Output

1. 1.Console Log OutPut
2. 2.File Log OutPut

1.Console Log Output

By default, Spring Boot logs messages to the console using Logback. This is useful during development for real-time feedback.

Example:

2025-10-25 10:30:45.123 INFO 12345 --- [main] com.example.MyApp : Application started

2. File Log Output

You can configure Spring Boot to write logs to a **file** for persistence, auditing, or production monitoring.

Configuration (application.properties):

```
# Log file name and location
logging.file.name=app.log
# OR specify a directory
logging.file.path=logs

# Optional: Customize file log format
logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
```

Log Levels

Log levels define the severity or importance of a log message. Spring Boot (via Logback or other logging frameworks) supports standard levels to help developers filter and manage logs effectively.Spring Boot supports all logger levels such as "TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL", "OFF". You can define Root logger in the application.properties file as shown below

```
logging.level.root = WARN
```

EXCEPTION HANDLING

Exception handling is the process of catching and responding to errors that occur during the execution of a Spring Boot application. It ensures that the application can gracefully handle unexpected situations and provide meaningful feedback to users or clients.

Spring Boot provides several ways to handle exceptions:

- Globally
- Per controller
- Using custom exception classes

1.Using @ExceptionHandler in Controller

@ExceptionHandler is used within a controller to handle specific exceptions locally. It allows you to return a custom response when a particular exception is thrown in that controller.

```java
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable int id) {
        if (id <= 0) {
            throw new IllegalArgumentException("User ID must be positive");
        }
        return new User(id, "John Doe");
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleIllegalArgument(IllegalArgumentException ex) {
        return ResponseEntity.badRequest().body("Error: " + ex.getMessage());
    }
}
```

USE WHEN:

- ❖ You want to handle exceptions specific to a single controller.
- ❖ You need custom logic for different exceptions in different controllers.

2. Using @ControllerAdvice for Global Handling

@ControllerAdvice is used to define global exception handlers that apply to all controllers. It helps centralize error handling logic in one place.

```java
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleAllExceptions(Exception ex) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body("Something went wrong: " + ex.getMessage());
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleNotFound(ResourceNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                             .body("Not Found: " + ex.getMessage());
    }
}
```

USE WHEN:

- ❖ You want to handle exceptions globally across all controllers.
- ❖ You want to standardize error responses for your REST API.

3.Custom Exception Class

Creating custom exceptions allows you to define domain-specific errors and handle them cleanly using @ExceptionHandler or @ControllerAdvice.

```java
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

USAGE IN CONTROLLER:

```java
@GetMapping("/{id}")
public User getUser(@PathVariable int id) {
    return userRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("User not found with ID: " + id));
}

```

USE WHEN:

- ❖ You want to represent specific error conditions in your domain.
- ❖ You want to improve code readability and maintainability.

INTERCEPTOR

An interceptor in Spring Boot is a component that allows you to intercept HTTP requests and responses before they reach the controller or after the controller processes them. It is part of the Spring MVC framework and is commonly used for:

- ❖ Logging
- ❖ Authentication/authorization
- ❖ Request modification
- ❖ Response transformation
- ❖ Performance monitoring

Interceptor Methods

- ❖ **preHandle()** method – This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.
- ❖ **postHandle()** method – This is used to perform operations before sending the response to the client.
- ❖ **afterCompletion()** method – This is used to perform operations after completing the request and response.

Pros of Using Interceptors

- ❖ Separation of Concerns: Cleanly separates cross-cutting concerns like logging, security, etc.
- ❖ Reusable: Can be applied across multiple controllers or endpoints.
- ❖ Flexible: Can be configured for specific URL patterns.

Cons of Using Interceptors

- Complexity: Overuse can make debugging harder.
- Limited to HTTP Layer: Not suitable for business logic or service-level concerns (use AOP for that)
- Order Matters: Multiple interceptors require careful ordering.

SERVLET FILTER

A Servlet Filter is a Java component that allows you to intercept and modify HTTP requests and responses before they reach a servlet or after the servlet processes them. Filters are part of the Servlet API and are often used in Spring Boot for:

- Logging
- Authentication and authorization
- Request/response modification
- Compression
- CORS handling

Filter Lifecycle

- **Initialization**: init(FilterConfig config)
  - Called once when the filter is created.
- **Filtering**: doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
  - Called for every request/response.
  - Must call chain.doFilter() to pass control.
- **Destruction**: destroy()
  - Called when the filter is removed or app shuts down.

Key Differences Table

Aspect	Servlet Filter	Interceptor
Layer	Web layer (Servlet)	Controller layer (Spring MVC)
API	Java Servlet API	Spring MVC API
Configuration	`web.xml` / `@WebFilter`	`WebMvcConfigurer` / Spring config
Methods	`init()`, `doFilter()`, `destroy()`	`preHandle()`, `postHandle()`, `afterCompletion()`
Framework	Works without Spring	Requires Spring MVC
Use Cases	Authentication, logging, compression	Controller-specific logic, adding attributes

Execution Flow in Spring MVC:

CLIENT → FILTER → DISPATCHERSERVLET → INTERCEPTOR → CONTROLLER → VIEW

CORS SUPPORT

Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.

For example, your web application is running on 8080 port and by using JavaScript you are trying to be consuming RESTful web services from 9090 port. Under such situations, you will face the Cross-Origin Resource Sharing security issue on your web browsers.

Two requirements are needed to handle this issue

- ❖ RESTful web services should support the Cross-Origin Resource Sharing.
- ❖ RESTful web service application should allow accessing the API(s) from the 8080 port.

Enable CORS in Controller Method

We need to set the origins for RESTful web service by using **@CrossOrigin** annotation for the controller method. This @CrossOrigin annotation supports specific REST API, and not for the entire application.

```
@RequestMapping(value = "/products")
@CrossOrigin(origins = "http://localhost:8080")
public ResponseEntity<Object> getProduct() {
   return null;
}
```

Global CORS Configuration

We need to define the shown @Bean configuration to set the CORS configuration support globally to your Spring Boot application.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
   return new WebMvcConfigurer() {
      @Override
      public void addCorsMappings(CorsRegistry registry) {
         registry.addMapping("/products").allowedOrigins("http://localhost:9000");
      }
   };
}
```

INTERNATIONALIZATION

Internationalization is the process of designing your application so it can easily support multiple languages and cultural formats without requiring code changes. It focuses on making the app flexible enough to adapt to different locales, such as English (US), French (France), or Japanese (Japan). This is essential for global applications because users expect content in their native language and proper formatting for dates, numbers, and currencies.

Difference Between Internationalization and Localization

- Internationalization (i18n): Prepares the app for multiple languages and regions.
- Localization (l10n): Implements translations and cultural adjustments for a specific locale.

Core Components

Locale

A locale defines language and region settings, such as en-US or fr-FR. It influences text, date, number, and currency formats.

Resource Bundles

These are files that store translations for different languages.

- messages_en.properties
- messages_fr.properties

Formatting

Dates, numbers, and currencies should be formatted according to the user's locale. For instance, 1,000.50 in US vs 1.000,50 in France.

Character Encoding

Always use UTF-8 to support all languages, including those with special characters like Chinese or Arabic.

SCHEDULING

Scheduling means running tasks automatically at specific times or intervals without manual triggers. In Spring Boot, you typically use the @Scheduled annotation to run jobs like cleanup tasks, email reminders, report generation, cache refreshes, or data syncs. Scheduling is essential for automation and reliability in backend systems and microservices.

- Main point to remember: Scheduling helps you automate recurring work—make jobs idempotent, observable, and resilient.
- Spring provides a simple way to enable scheduling at the application level via @EnableScheduling. You can place this on a configuration class or your main application class.

Scheduling Options: fixedRate, fixedDelay, cron

1) fixedRate

Runs the method at a **fixed interval**, regardless of how long the method takes (it schedules based on start time).

```java
@Component
public class FixedRateJob {
    @Scheduled(fixedRate = 5000) // every 5 seconds
    public void run() {
        // Logic
    }
}
```

Use when: You want periodic tasks to start at regular intervals—even if the previous execution is still running (be careful with concurrency).

2) fixedDelay

Runs the method after a delay from the end of the previous run.

```java
@Component
public class FixedDelayJob {
    @Scheduled(fixedDelay = 5000) // 5s after previous completion
    public void run() {
        // Logic
    }
}
``
```

Use when: Tasks must **not overlap** and should wait until the previous execution finishes.

3) cron

Runs on a calendar-based schedule using cron expressions.

```java
@Component
public class CronJob {
    // Every day at 02:30 AM Asia/Kolkata
    @Scheduled(cron = "0 30 2 * * *", zone = "Asia/Kolkata")
    public void run() {
        // Logic
    }
}
```

COMMON CRON EXAMPLES:

- Every minute: 0 * * * * *
- Every hour at :00: 0 0 * * * *
- Every weekday at 9:15: 0 15 9 * * MON-FRI

Main point to remember: Prefer cron for business schedules; use zone to avoid time zone drift (e.g., India Standard Time).

SENDING EMAIL VIA SMTP

In Spring Boot, sending emails via SMTP is a common feature used for notifications, password resets, alerts, and user communication. Spring Boot simplifies this process using the spring-boot-starter-mail dependency, which integrates JavaMailSender for composing and sending emails. SMTP (Simple Mail Transfer Protocol) is the standard protocol for email transmission across the internet. You can configure Spring Boot to use any SMTP provider like Gmail, Outlook, SendGrid, or corporate mail servers.

The process involves setting up SMTP properties in the application.properties file, creating a service class to send emails, and optionally using HTML formatting or attachments. For Gmail, you must use an App Password due to enhanced security.

ENABLING HTTPS

Enabling HTTPS in Spring Boot means securing your application with SSL/TLS so that all communication between client and server is encrypted. Here's a complete guide with explanations under each heading and key points to remember.

Why Enable HTTPS?

HTTPS ensures:

- Data encryption: Protects sensitive information like passwords and tokens.
- Authentication: Confirms the server identity using certificates.
- Integrity: Prevents data tampering during transmission.

Main point: HTTPS is critical for security and compliance in production environments.

You need to follow the steps given below to configure the HTTPS and the port 443 in Spring Boot application –

- Obtain the SSL certificate Create a self-signed certificate or get one from a Certificate Authority
- Enable HTTPS and 443 port

Self-Signed Certificate

To create a self-signed certificate, Java Run Time environment comes bundled with certificate management utility key tool. This utility tool is used to create a Self-Signed certificate. It is shown in the code given here.

```
keytool -genkey -alias tomcat -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore keystore.p12 -validity 3650
Enter keystore password:
   Re-enter new password:
   What is your first and last name?
   [Unknown]:
   What is the name of your organizational unit?
   [Unknown]:
   What is the name of your organization?
   [Unknown]:
   What is the name of your City or Locality?
   [Unknown]:
   What is the name of your State or Province?
   [Unknown]:
   What is the two-letter country code for this unit?
   [Unknown]:
   Is CN = Unknown, OU=Unknown, O = Unknown, L = Unknown, ST = Unknown, C = Unknown correct?
   [no]: yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 3,650 days
        for: CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
```

This code will generate a PKCS12 keystore file named as **keystore.p12** and the certificate alias name is tomcat. We've stored the keystore in E:/ > Dev directory and keystore password is springboot.

Configure HTTPS:

We need to provide the server port as 443, key-store file path, key-store-password, key-store-type and key alias name into the application.properties file.

```
server.port: 443
server.ssl.key-store: E:/Dev/keystore.p12
server.ssl.key-store-password: springboot
server.ssl.keyStoreType: PKCS12
server.ssl.keyAlias: tomcat
```

EUREKA SERVER

1) Eureka Server

A Spring Cloud Netflix service registry. It stores and serves the list of running service instances so that microservices can discover each other by name instead of static host/port.
Remember: Central directory for service discovery. Typically runs on port 8761 with the web dashboard.

2) Eureka Client

Any microservice that registers itself with the Eureka Server and (optionally) fetches the registry to discover others.

Key behaviors:

- Sends periodic heartbeats to renew its lease.
- Registers under spring.application.name (the service name).
- Can call peers by name via Spring Cloud LoadBalancer (@LoadBalanced RestTemplate/WebClient).

3) Service Registry

The in-memory (replicated) store held by Eureka Server containing:

- Service name → list of instances (IP/host, port, status)
- Metadata (version, zone, tags)
- Timestamps & lease info
- Use: Discovery clients query it to resolve URLs dynamically.

4) Instance

A single running copy of a service (e.g., orders-service running on 10.0.1.23:8081). Multiple instances under the same service name provide load balancing and fault tolerance.
Tip: Stabilize instance IDs (e.g., include hostname/port) for easier debugging.

5) Heartbeat

A periodic renewal call from client → server (default every 30s or tuned) indicating liveness. Missing heartbeats eventually trigger eviction.
Why it matters: Prevents stale entries in the registry during failures or redeploys.

6) Lease

The time-bound registration contract between a client and the server.
Key configs:

- lease-renewal-interval-in-seconds (heartbeat frequency)
- lease-expiration-duration-in-seconds (eviction timeout)
- Trade-off: Faster expiry = quicker failover but more sensitivity to transient network issues.

7) Self-Preservation Mode

Safety mechanism on Eureka Server that slows/halts evictions when many heartbeats are simultaneously missed (likely a network partition), avoiding a catastrophic "empty registry."
Best practice: Keep ON in production; tune renewal-percent-threshold thoughtfully.

8) Eviction

Removal of instances that haven't renewed their lease within the expiration window. Runs at an interval (default ~60s).
Symptoms to watch: Flapping UP/DOWN in the dashboard → usually indicates health/lease misconfiguration or slow health endpoints.

9) Default Zone

The Eureka endpoint clients use to register/fetch:

```
eureka.client.service-url.defaultZone: http://localhost:8761/eureka/
```

In HA, point to multiple peer servers or a load-balanced URL.

10) Metadata Map

Custom key-value pairs attached to an instance, e.g., version=v2, commit=abc123, region=apac. Used for routing, blue/green, or feature gates.

11) DiscoveryClient

The abstraction to query the registry programmatically:

```
List<ServiceInstance> instances = discoveryClient.getInstances("orders-service");
```

Useful for custom routing and health dashboards.

12) Load Balancer (Client-side)

Modern Spring Cloud uses Spring Cloud LoadBalancer (Ribbon is deprecated). With @LoadBalanced, you call http://service-name/... and the client picks an instance from Eureka (round-robin by default, pluggable strategies).

13) Peer Awareness / HA

Run 2–3 Eureka Servers as peers. Each server registers with and fetches from the others, replicating the registry for high availability.
Pattern: Profiles a/b/c with cross-referenced defaultZone URLs.

14) Zones & Regions

Logical grouping for fault isolation or geo-routing. Clients can prefer same-zone instances to reduce latency and cross-AZ traffic.

15) Health Integration

Enable Spring Boot Actuator so Eureka uses /actuator/health to determine UP/DOWN. Slow or failing health checks cause instance instability.

16) Security

Protect the Eureka dashboard and APIs:

- Basic Auth / OAuth for clients
- HTTPS/TLS via reverse proxy or Spring Boot SSL
- Restrict ingress; don't expose the registry publicly

Common Pitfalls

- Version mismatch between Spring Boot and Spring Cloud BOM → discovery failures.
- Wrong defaultZone URL (missing /eureka/) → not registering.
- Heartbeat/lease too aggressive → frequent evictions.
- Docker/K8s networking misconfig (hostnames/IPs) → unreachable instances.
- Registry "empty" → check firewall, DNS, NTP time sync.

GATEWAY PROXY SERVER

A Gateway Proxy Server acts as a single-entry point for all client requests in a microservices architecture. Instead of clients calling services directly, they route through the gateway, which handles:

- Routing requests to appropriate services.
- Cross-cutting concerns like authentication, logging, rate limiting.
- Load balancing and service discovery integration.
- **Main point:** It simplifies client interaction and centralizes security and monitoring.

SPRING CLOUD CONFIG SERVER

Spring Cloud Config Server is a centralized configuration management solution for distributed systems. Instead of hardcoding properties in each microservice, you store them in a central repository (usually Git). The Config Server serves these properties to clients at runtime, ensuring consistency and easy updates across environments.

Main point: It solves the problem of managing configuration for multiple services in microservices architecture.

Config Repository Structure

```
application.yml              # Global defaults
orders-service.yml           # Service-specific config
inventory-service.yml        # Another service
orders-service-dev.yml       # Profile-specific config
```

NAMING CONVENTION:

{application}-{profile}.yml or .properties.

SPRING CLOUD CONFIG CLIENT

A Config Client is any Spring Boot application that retrieves its configuration from a Spring Cloud Config Server instead of local property files. This allows centralized management of configuration for multiple microservices, ensuring consistency and easy updates across environments.

Main point: Config Client fetches properties during the bootstrap phase from the Config Server.

ACTUATOR

Spring Boot Actuator is a set of production-ready features that help you monitor and manage your application. It exposes endpoints for health checks, metrics, environment info, and more, making it easier to integrate with monitoring tools like Prometheus, Grafana, or ELK.

Main point: Actuator gives visibility into your app's internals without writing custom monitoring code.

ENDPOINTS	USAGE
/metrics	To view the application metrics such as memory used, memory free, threads, classes, system uptime etc.
/env	To view the list of Environment variables used in the application.
/beans	To view the Spring beans and its types, scopes and dependency.
/health	To view the application health
/info	To view the information about the Spring Boot application.
/trace	To view the list of Traces of your Rest endpoints.

Adding Actuator to Your Project

Add dependency in pom.xml:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

SPRING BOOT ADMIN SERVER

Spring Boot Admin Server is a monitoring and management tool for Spring Boot applications. It provides a web UI to visualize the status of multiple services, their health, metrics, environment properties, and logs. It's widely used in microservices architectures to centralize monitoring.

Main point: It aggregates Actuator endpoints from multiple Spring Boot apps into a single dashboard.

ADMIN CLIENT

A Spring Boot Admin Client is any Spring Boot application that exposes Actuator endpoints and registers with the Spring Boot Admin Server. Once registered, the Admin Server can display the app's health, metrics, environment, logs, threads, mappings, and more—across all your microservices. Registration can be direct (via the admin client starter) or indirect (via service discovery like Eureka; the Admin Server discovers you and connects to your Actuator endpoints).

Main point: An Admin Client is just your regular Boot app + Actuator + a way for the Admin Server to find and authorize it.

Architecture Overview

EUREKA SERVER – service registry (directory of all services).
ADMIN SERVER – web UI that discovers services via Eureka and aggregates their Actuator data (health, metrics, logs, etc.).
ADMIN CLIENTS – your Spring Boot apps registered with Eureka and exposing Actuator endpoints.

Flow:
Clients ⟶ register in Eureka ⟶ Admin Server queries Eureka, finds clients, and pulls their Actuator endpoints to display in the UI.

Main point: With Eureka integration, you don't need to manually register each client to Admin Server; discovery is automatic.

SWAGGER

Swagger is an open-source framework that helps developers design, build, document, and consume RESTful APIs. It provides a standardized way to describe APIs using the OpenAPI specification and offers tools like Swagger UI for interactive documentation. With Swagger, developers can easily understand API endpoints, request/response formats, and test them without writing extra code.
Key point: Swagger makes APIs self-descriptive and improves developer experience.

WHY ENABLE SWAGGER IN SPRING BOOT?

Modern applications often expose multiple REST endpoints. Without proper documentation, it becomes hard for teams and external consumers to understand and use these APIs. Swagger solves this by:

- ❖ Auto-generating API documentation from Spring controllers.
- ❖ Providing an interactive UI for testing endpoints.
- ❖ Supporting client SDK generation for multiple languages.
- ❖ Ensuring consistency across microservices.

Key point: Swagger reduces manual documentation effort and improves API usability.

SPRINGDOC OPENAPI

SpringDoc OpenAPI is a modern library that integrates OpenAPI 3 specification with Spring Boot applications. It automatically generates API documentation from your REST controllers and provides an interactive Swagger UI for testing endpoints. Unlike older Swagger2 implementations (Springfox), SpringDoc is actively maintained and fully compatible with Spring Boot 2.x and 3.x.
Key point: SpringDoc is the recommended approach for API documentation in Spring Boot projects today.

WHY USE SPRINGDOC OPENAPI?

- OpenAPI 3 support: Latest standard for API documentation.
- Zero boilerplate: Minimal configuration required.
- Interactive UI: Built-in Swagger UI for testing endpoints.
- Integration with Spring Boot: Works seamlessly with Actuator, Security, and WebFlux.
- Customizable: Supports grouping, filtering, and advanced annotations.

Key point: It simplifies API documentation while adhering to modern standards.

TRACING MICRO SERVICE LOGS

Log tracing refers to the ability to track a single request across multiple microservices by correlating logs. In a microservices architecture, a single user action often triggers calls to multiple services. Without tracing, debugging issues becomes extremely difficult because logs are scattered across different services.

Key point: Tracing connects logs from different services for the same request using a unique identifier.

WHY IS TRACING IMPORTANT?

- Debugging: Quickly identify where a request failed.
- Performance Monitoring: Detect bottlenecks across services.
- Root Cause Analysis: Understand the flow of requests.
- Compliance: Maintain audit trails for sensitive operations.

Key point: Tracing improves reliability and reduces MTTR (Mean Time to Resolution).

Spring Cloud Sleuth:

Spring Cloud Sleuth is a distributed tracing library for Spring Boot applications. It automatically adds trace IDs and span IDs to logs, enabling developers to track a request across multiple microservices. Sleuth integrates seamlessly with Zipkin, Jaeger, and OpenTelemetry, providing visibility into request flows and latency.

Key point: Sleuth makes tracing easy by adding correlation IDs without manual coding.

WHY USE SLEUTH?

In microservices, a single request often spans multiple services. Without tracing, debugging issues is difficult because logs are scattered. Sleuth solves this by:

- Adding traceId and spanId to every log.
- Propagating these IDs across HTTP calls, messaging queues, and async tasks.
- Integrating with distributed tracing systems for visualization.

Key point: Sleuth improves observability and reduces troubleshooting time.

HOW SLEUTH WORKS

- When a request enters a service, Sleuth creates a trace and a span.
- It adds IDs to logs using MDC (Mapped Diagnostic Context).
- When calling another service, Sleuth propagates these IDs in headers.
- The next service continues the trace, creating new spans.

```
2025-10-26 09:18:06 INFO [orders-service,traceId=abc123,spanId=def456] Order created
```

- **Application-name** – Name of the application
- **Traceid** – each request and response traceid is same when calling same service or one service to another service.
- **Spanid** – Span Id is printed along with Trace Id. Span Id is different every request and response calling one service to another service.
- **Zipkin-export** – By default it is false. If it is true, logs will be exported to the Zipkin server.

FILE HANDLING

File Handling in Java allows programs to interact with files stored on disk—reading data from files, writing data to files, and performing operations like copying, deleting, or checking file properties. Java provides the java.io and java.nio.file packages for these operations. File handling is crucial in backend systems, logging, configuration management, and data processing.

Implementation of the Application

Step 1: Setting up the Application.Properties file with configurations required for multipart file upload.

```
spring.servlet.multipart.enabled=true
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=10MB
```

These configurations can be explained as follows:

spring.servlet.multipart.enabled - determines whether multipart has to be enabled or not

spring.servlet.multipart.max-file - specifies the maximum size of the file allowed for uploading.

spring.servlet.multipart.max-request-size - specifies the maximum size allowed for multipart/form-data requests.

Step 2: Create a RestController FileController that handles the following REST APIs:

1. Upload API

Usage: It can be used to upload a file. It consumes Multipart requests.
URL: /upload

HttpMethod: POST

Implementation Details:

For developing this API, we are using MultipartFile as a Request Parameter. The uploaded file is sent as form data which is then retrieved in the Rest controller as a Multipart file. So, MultipartFile is nothing but a representation of an uploaded file received in a multipart request.

2. getFiles API

Usage: It can be used to get a list of filenames that have been uploaded.
URL: /getFIles
HttpMethod: GET

Implementation Details:

It can be implemented by simply using the list() method of java.io.File which returns an array of strings naming the files and directories in the directory denoted by the given abstract pathname.

3. Download API

It can be used to download a previously uploaded file.
URL: /download/{filename}
HttpMethod: POST

Implementation Details:

To Implement this API, we first check whether the requested file for download exists or not in the uploaded folder. If the file exists, we use InputStreamResource to download the file. It is also required to set Content-Disposition in the response header as an attachment and MediaType as application/octet-stream.

CACHING

A Cache is any temporary storage location that lies between the application and persistence database or a third-party application that stores the most frequently or recently accessed data so that future requests for that data can be served faster. It increases data retrieval performance by reducing the need to access the underlying slower storage layer. Data access from memory is always faster in comparison to fetching data from the database. Caching keeps frequently accessed objects, images, and data closer to where you need them, speeding up access by not hitting the database or any third-party application multiple times for the same data and saving monetary costs. Data that does not change frequently can be cached.

Types of Caching

There are mainly 4 types of Caching :

- CDN Caching
- Database Caching
- In-Memory Caching
- Web server Caching

1. CDN Caching

A content delivery network (CDN) is a group of distributed servers that speed up the delivery of web content by bringing it closer to where users are. Data centres across the globe use caching, to deliver internet content to a web-enabled device or browser more quickly through a server near you, reducing the load on an application origin and improving the user experience. CDNs cache content like web pages, images, and video in proxy servers near your physical location.

2. Database Caching

Database caching improves scalability by distributing query workload from the backend to multiple front-end systems. It allows flexibility in the processing of data. It can significantly reduce latency and increase throughput for read-heavy application workloads by avoiding, querying a database too much.

3. In-Memory Caching

In-Memory Caching increases the performance of the application. An in-memory cache is a common query store, therefore, relieves databases of reading workloads. Redis cache is one of the examples of an in-memory cache. Redis is distributed, and advanced caching tool that allows backup and restores facilities. In-memory Cache provides query functionality on top of caching.

4. Web server Caching

Web server caching stores data, such as a copy of a web page served by a web server. It is cached or stored the first time a user visits the page and when the next time a user requests the same page, the content will be delivered from the cache, which helps keep the origin server from getting overloaded. It enhances page delivery speed significantly and reduces the work needed to be done by the backend server.

Cache Annotations of Spring Boot

1. @EnableCaching

@EnableCaching is the starting point for enabling Spring's caching mechanism. Without this annotation, none of the cache-related annotations will work because the caching infrastructure won't be activated. You typically place it on your main application class or any configuration class. It registers the necessary beans like CacheInterceptor and sets up proxy-based method interception.

```java
@SpringBootApplication
@EnableCaching
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

This ensures that all caching annotations (@Cacheable, @CachePut, @CacheEvict) will function properly.

2. @Cacheable

@Cacheable is used on methods whose results should be cached. When the method is called for the first time with a given key, the result is computed and stored in the cache. Subsequent calls with the same key return the cached value without executing the method again. You can customize the cache name, key, and conditions using attributes like cacheNames, key, condition, and unless.

```java
@Service
public class ProductService {
    @Cacheable(cacheNames = "products", key = "#id", unless = "#result == null")
    public Product getProductById(Long id) {
        System.out.println("Fetching from DB...");
        return productRepository.findById(id).orElse(null);
    }
}
```

Here, the product is fetched from the database only once per ID. If the result is null, it won't be cached.

3. @CachePut

@CachePut always executes the method and updates the cache with the result. It's useful for update operations where you want the cache to reflect the latest state immediately after saving data. Unlike @Cacheable, it does not skip method execution.

```java
@Service
public class ProductService {
    @CachePut(cacheNames = "products", key = "#product.id")
    public Product updateProduct(Product product) {
        return productRepository.save(product);
    }
}
```

This ensures that after updating a product, the cache entry for that product ID is refreshed.

4. @CacheEvict

@CacheEvict removes entries from the cache. It's commonly used in delete operations or when data changes invalidate cached results. You can evict a single-entry using key or clear the entire cache using allEntries = true. The beforeInvocation attribute allows eviction before the method runs.

```java
@Service
public class ProductService {
    @CacheEvict(cacheNames = "products", key = "#id")
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

    @CacheEvict(cacheNames = "products", allEntries = true)
    public void clearCache() {
        System.out.println("All product cache entries cleared.");
    }
}
```

5. @Caching

@Caching is a composite annotation that allows multiple cache operations on a single method. It's useful when you need to update one cache and evict another simultaneously.

```java
@Service
public class UserService {
    @Caching(
        put = { @CachePut(value = "users", key = "#user.id") },
        evict = { @CacheEvict(value = "userList", allEntries = true) }
    )
    public User saveUser(User user) {
        return userRepository.save(user);
    }
}
```

Here, the individual user cache is updated, and the user list cache is cleared to maintain consistency.

6. @CacheConfig

@CacheConfig is applied at the class level to define common cache settings for all methods in that class. It reduces duplication by setting defaults like cacheNames and keyGenerator.

```java
@Service
@CacheConfig(cacheNames = "products")
public class ProductService {
    @Cacheable(key = "#id")
    public Product getProduct(Long id) {
        return productRepository.findById(id).orElse(null);
    }

    @CachePut(key = "#product.id")
    public Product saveProduct(Product product) {
        return productRepository.save(product);
    }
}
```

CACHE PROVIDER

In **Spring Boot**, a **cache provider** is the underlying implementation that actually stores and retrieves cached data when you use Spring's caching abstraction. Spring Boot itself doesn't implement caching logic; instead, it provides an abstraction layer (@EnableCaching, @Cacheable, @CachePut, @CacheEvict) and delegates the actual caching to a provider.

A CACHE PROVIDER IS A LIBRARY OR FRAMEWORK THAT MANAGES THE CACHE STORAGE AND RETRIEVAL. SPRING BOOT SUPPORTS MULTIPLE PROVIDERS THROUGH ITS CACHING ABSTRACTION.

Spring boot cache providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the **org.springframework.cache.Cache** and **org.springframework.cache.CacheManager** interfaces .If we have not defined a bean of type **CacheManager** or a **CacheResolver** named **cacheResolver**, Spring Boot tries to detect the following providers:

- **Generic**
- **JCache (JSR-107)**
- **EhCache 2.x**
- **Hazelcast**
- **Guava**
- **Infinispan**
- **Couchbase**
- **Redis**
- **Caffeine**
- **Simple**

1. Generic (Spring Cache Abstraction)

Spring provides a generic caching abstraction via annotations like @Cacheable, @CachePut, and @CacheEvict. This abstraction allows you to switch between different cache providers without changing your business logic.

```java
@EnableCaching
@SpringBootApplication
public class App {}
```

```java
@Cacheable("products")
public Product getProductById(Long id) {
    return productRepository.findById(id).orElseThrow();
}
```

No provider-specific code here; Spring delegates to the configured CacheManager.

2. JCache (JSR-107)

JCache is a Java standard API for caching. It defines interfaces like CacheManager, Cache, and annotations similar to Spring's. Providers like Ehcache 3.x and Hazelcast implement JCache.

Add dependency:

```xml
<dependency>
    <groupId>javax.cache</groupId>
    <artifactId>cache-api</artifactId>
</dependency>

```

Enable JCache:

```
spring.cache.type=jcache
spring.cache.jcache.config=classpath:ehcache.xml
```

3. EhCache 2.x

Ehcache is a **popular in-memory cache** with disk persistence and clustering support. Version 2.x is older but still widely used.

Add dependency:

```xml
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
```

Config:

```
spring.cache.type=ehcache
spring.cache.ehcache.config=classpath:ehcache.xml
```

4. Hazelcast

Hazelcast is an **in-memory data grid** for distributed caching. Ideal for microservices or clustered environments.

Add dependency:

```xml
<dependency>
    <groupId>com.hazelcast</groupId>
    <artifactId>hazelcast-spring</artifactId>
</dependency>
```

Config:

```
spring.cache.type=hazelcast
```

Hazelcast auto-configures a cluster if multiple nodes run.

5. Guava

Guava provides a **lightweight local cache** using CacheBuilder. Good for small-scale apps.

Add dependency:

```xml
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
</dependency>
```

Config:

```
spring.cache.type=guava
spring.cache.guava.spec=maximumSize=500,expireAfterAccess=600s
```

6. Infinispan

Infinispan is a **distributed in-memory key/value store** with advanced features like transactions and persistence.

Add dependency:

```xml
<dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-spring-boot-starter</artifactId>
</dependency>
```

Config:

```
spring.cache.type=infinispan
```

7. Couchbase

Couchbase is a NoSQL database that can act as a cache layer. Useful for hybrid persistence + caching.

Add dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-couchbase</artifactId>
</dependency>
```

Config:

```
spring.cache.type=couchbase
spring.couchbase.connection-string=localhost
spring.couchbase.username=admin
spring.couchbase.password=password
```

8. Redis

Redis is a distributed in-memory data store widely used for caching and pub/sub.

Add dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Config:

```
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

9. Simple

Simple cache uses ConcurrentHashMap internally. It's the default fallback if no provider is configured.

```
spring.cache.type=simple
```

No extra setup required.

10. Caffeine

Caffeine is a high-performance local cache with advanced eviction policies and async loading.

Add dependency:

```xml
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
</dependency>
```

Config:

```
spring.cache.type=caffeine
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

UNIT TESTING

Unit Testing is a software testing technique where individual components or functions of a program—called *units*—are tested in isolation to ensure they work as expected.

Junit

JUnit is a popular **unit testing framework for Java** that helps developers write and run tests for their code. It is widely used because it is lightweight, easy to integrate, and supports automated testing. JUnit ensures that individual units of code (like methods or classes) work as expected, which improves software reliability and reduces bugs.

Why JUnit?

- **Automation:** JUnit supports automated testing, allowing developers to quickly verify that recent changes haven't disrupted existing functionality. JUnit tests can be integrated into build tools like Maven or Gradle and CI/CD pipelines.
- **Early Bug Detection**: Unit tests help catch bugs early in the development process, reducing the likelihood of introducing new errors as code evolves.
- **Maintainability**: JUnit ensures that changes, bug fixes, or new features don't inadvertently break other parts of the system. This helps maintain the overall quality of the codebase.
- **Ease of Use**: With simple annotations and methods, JUnit is easy to set up and use, making it accessible even for developers who are new to testing.

Basic Concepts in JUnit

1. Test Case: A test case is a single unit test written to check a specific piece of code. In JUnit, a test case is typically a method within a class.

2. Annotations:

- @Test: Marks a method as a test case.
- @Before: Executes the code before each test method.
- @After: Executes the code after each test method.
- @BeforeClass and @AfterClass: Executed once before and after all tests in a class, used for setup and cleanup.

3. Assertions:

JUnit provides several assertions like assertEquals(), assertTrue(), and assertNotNull() to verify that the expected results match the actual results.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {

    @Test
    void testAddition() {
        int result = 2 + 3;
        assertEquals(5, result, "Addition should return correct sum");
    }
}
```

JUnit in the Application Development Cycle

JUnit plays a crucial role in modern software development by enabling automated testing at different stages of the development process. It ensures that code changes do not break existing functionality and helps maintain high-quality standards throughout the lifecycle.

1. Test-Driven Development (TDD)

In TDD, developers write tests before writing the actual implementation code. This approach ensures that every piece of functionality is clearly defined and verified early. JUnit is the backbone of TDD in Java projects because:

- Developers create unit tests using JUnit annotations like @Test.
- These tests fail initially (since the code isn't implemented yet).
- Developers then write the minimum code required to make the test pass.
- This cycle of Red → Green → Refactor continues until the feature is complete.

Benefits of TDD with JUnit:

- Reduces bugs by validating logic early.
- Encourages modular and maintainable code.
- Provides confidence during refactoring.

2. Continuous Integration (CI)

JUnit integrates seamlessly with CI/CD pipelines (e.g., Jenkins, GitHub Actions, GitLab CI). In this setup:

- Every time a developer commits code, the CI system automatically runs JUnit tests.
- If tests fail, the build is marked as broken, preventing faulty code from being deployed.
- This automation ensures fast feedback, early detection of issues, and consistent code quality.

Benefits of CI with JUnit:

- Eliminates manual testing overhead.
- Supports agile development and frequent releases.
- Improves collaboration among teams by catching errors before merging.

MOCKITO

Mockito is a mocking framework for Java that works alongside JUnit to create mock objects for unit testing. A mock object is a simulated version of a real object that mimics its behavior without requiring the actual implementation. This is especially useful when:

- The real object is complex or depends on external systems (e.g., databases, APIs).
- You want to isolate the unit under test and avoid side effects.

Why Use Mockito

- Isolated Testing: Test classes without relying on real dependencies.
- Faster Tests: Avoid slow operations like database or network calls.
- Cleaner Code: Reduce boilerplate with annotations like @Mock and @InjectMocks.
- Behavior Verification: Easily check if methods were called with expected arguments.
- Seamless Framework Support: Works seamlessly with JUnit 4, JUnit 5 and Testing.

CORE CONCEPTS OF MOCKITO

Mockito is a popular Java framework used for creating mock objects in unit tests. It works alongside JUnit to isolate the unit under test by simulating dependencies. Below are the core concepts of Mockito explained in detail.

1. Mock Objects

Mock objects are fake implementations of real objects that simulate their behaviour for testing purposes. They allow you to test a class in isolation by replacing its dependencies with controlled, predictable versions.

2. Stubbing

Stubbing is the process of defining what a mock object should return when a specific method is called. It's used to simulate different scenarios such as returning fixed values, throwing exceptions, or mimicking slow responses.

3. Verification

Verification is the process of checking whether certain methods were called on mock objects during test execution. It ensures that the tested code interacts with its dependencies as expected like calling a method a specific number of times with given arguments.

The problem that Mockito solved

Consider a scenario where you have a service class that depends on a database repository. Testing this service directly would require:

- Setting up a database connection
- Preparing test data
- Cleaning up after tests

Mockito annotations

@Mock

The @Mock annotation is used to create mock objects for classes or interfaces. These mock objects simulate the behaviour of real dependencies without executing actual logic. This is particularly useful when the real dependency interacts with external systems like databases or APIs, which you want to avoid during unit testing. By using @Mock, you can control the behaviour of these dependencies and make your tests faster and more reliable.

```
1  @Mock
2  private UserRepository userRepository;
```

@InjectMocks

The @InjectMocks annotation automatically injects the mocks created with @Mock into the class under test. This helps reduce boilerplate code and ensures that the class being tested receives its dependencies without manual setup.

```
1  @InjectMocks
2  private UserService userService;
```

@Spy

The @Spy annotation creates a partial mock, which means it uses a real object but allows you to override specific methods. This is useful when you want to test a class with its real behaviour but still control certain methods for predictable results.

```
1  @Spy
2  private UserService userService = new UserService();
```

@Captor

The @Captor annotation is used to capture arguments passed to mock methods. This is helpful when you need to verify what values were actually sent during method calls.

```
@Captor
ArgumentCaptor<String> stringCaptor;
```

@ExtendWith(MockitoExtension.class)

In JUnit 5, @ExtendWith(MockitoExtension.class) is required to enable Mockito annotations in your test class. It initializes mocks and handles dependency injection automatically:

```
@ExtendWith(MockitoExtension.class)
class MyTest { ... }
```